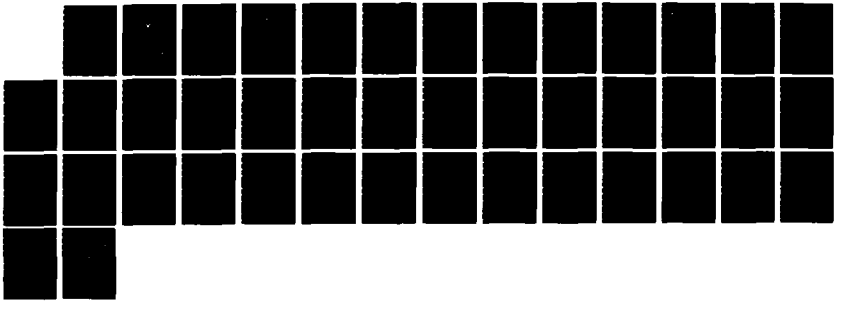
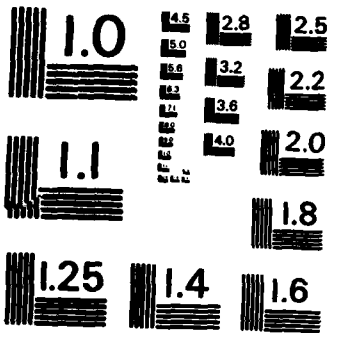
AD-A166 731 A PRIMER FOR PROGRAM MANAGERS; EMBEDDED SOFTWARE ACQUISITION(U) AIR COMMAND AND STAFF COLL MAXWELL AFB AL C C ALBERT APR 86 ACSC-86-0045 1/1

UNCLASSIFIED F/G 15/5 NL





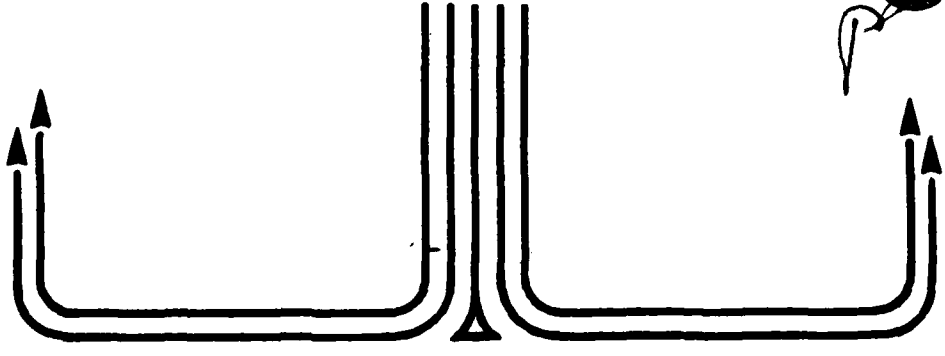
AIR COMMAND AND STAFF COLLEGE

STUDENT REPORT

A PRIMER FOR PROGRAM MANAGERS;
EMBEDDED SOFTWARE ACQUISITION

Major Cecilia C. Albert 86-0045

"insights into tomorrow"



DISCLAIMER

The views and conclusions expressed in this document are those of the author. They are not intended and should not be thought to represent official ideas, attitudes, or policies of any agency of the United States Government. The author has not had special access to official information or ideas and has employed only open-source material available to any writer on this subject.

This document is the property of the United States Government. It is available for distribution to the general public. A loan copy of the document may be obtained from the Air University Interlibrary Loan Service (AUL/LDEX, Maxwell AFB, Alabama, 36112) or the Defense Technical Information Center. Request must include the author's name and complete title of the study.

This document may be reproduced for use in other research reports or educational pursuits contingent upon the following stipulations:

-- Reproduction rights do not extend to any copyrighted material that may be contained in the research report.

-- All reproduced copies must contain the following credit line: "Reprinted by permission of the Air Command and Staff College."

-- All reproduced copies must contain the name(s) of the report's author(s).

-- If format modification is necessary to better serve the user's needs, adjustments may be made to this report--this authorization does not extend to copyrighted information or material. The following statement must accompany the modified document: "Adapted from Air Command and Staff Research Report (number) entitled (title) by (author) ."

-- This notice must be included with any reproduced or adapted portions of this document.

REPORT NUMBER 86-0045

TITLE A PRIMER FOR PROGRAM MANAGERS; EMBEDDED SOFTWARE ACQUISITION

AUTHOR(S) MAJOR CECILIA C. ALBERT, USAF

FACULTY ADVISOR MAJOR MANUEL T. TORRES, ACSC/EDOWA

SPONSOR LT COL ROBERT CHRISTOPHER, DSMC/SE

Submitted to the faculty in partial fulfillment of requirements for graduation.

AIR COMMAND AND STAFF COLLEGE
AIR UNIVERSITY
MAXWELL AFB, AL 36112

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

ADA 166 731

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT STATEMENT "A" Approved for public release; Distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 86-0045	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION ACSC/EDCC 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Maxwell AFB, AL 36112-5542	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) A PRIMER FOR PROGRAM MANAGERS;	
12. PERSONAL AUTHOR(S) Albert, Cecilia C., Major, USAF	
13a. TYPE OF REPORT 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1986 April 15. PAGE COUNT 39
16. SUPPLEMENTARY NOTATION ITEM 11: EMBEDDED SOFTWARE ACQUISITION (U)	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Computer Software has become an ever increasing element in today's major defense system acquisitions. This study was written for the program manager with no software experience who is faced with the prospect of acquiring a major defense system which contains embedded software. It provides an overview of the software development process and a discussion of why things go wrong. Both the software development itself and the implications of the software on the system are considered.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☒ DTIC USERS ☐ XX	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL ACSC/EDCC	22b. TELEPHONE NUMBER (Include Area Code) (205) 293-2483 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

PREFACE

Computer Software has become an ever increasing element in today's major defense system acquisitions. This study was written for the program manager with no software experience who is faced with the prospect of acquiring a major defense system which contains embedded software. It will provide an overview of the software development process and a discussion of why things go wrong. Both the software development itself and the implications of the software on the system will be considered.

This study was initiated by the recent publication of DoD Standard 2167. This standard establishes a uniform software development process for all aspects of the software system life cycle. It is hoped, that having read this paper, the program manager will be more comfortable with the role of software in system acquisition -- so comfortable the program manager will be prepared to actively consider software as a decision variable throughout the acquisition process.

ABOUT THE AUTHOR

The author was awarded her Bachelor of Liberal Arts, with a major in mathematics, from Sweet Briar College, Virginia, and received her commission from Officer's Training School, Lackland AFB, Texas, in 1972. She spent three years as an analytic and satellite command and control programer in Denver, Colorado, and Woomera, South Australia, before being selecting for an Air Force Institute of Technology assignment at Leland Stanford, Jr University, California, where she was awarded her Master of Computer Science Degree in 1977. She has since served in several space and computer science related fields: four years in payload operations in Sunnyvale, California, four years directing studies and simulation of new space technologies in Los Angeles, California, and two years as Systems Engineer for a multi-satellite mission planner in Baltimore, Maryland. She is currently assigned to Air Command and Staff College, Maxwell AFB, Alabama, with a follow-on assignment to Space Division, Los Angeles AFS, California.

TABLE OF CONTENTS

CONTINUED

LIST OF ILLUSTRATIONS

EXECUTIVE SUMMARY

Part of our College mission is distribution of the students' problem solving products to DoD sponsors and other interested agencies to enhance insight into contemporary, defense related issues. While the College has accepted this product as meeting academic requirements for graduation, the views and opinions expressed or implied are solely those of the author and should not be construed as carrying official sanction.

"insights into tomorrow"

REPORT NUMBER 86-0045

AUTHOR(S) MAJOR CECILIA C. ALBERT, USAF

TITLE A PRIMER FOR PROGRAM MANAGERS;
EMBEDDED SOFTWARE ACQUISITION

I. Background: Computer technology has progressed to the point that every system developed today contains an embedded computer. As our reliance on these computers increases, so too do the challanges for today's program managers. This study examines the current realities and illusions of embedded software development. It provides the program manager with some of the special factors which must be considered when embedded software is involved in the acquisition of a major system whose primary purpose is not data automation.

II. Problems: Software development is considered by many today as a "Black Art". This causes many of the problems encountered by program managers. Software is not considered early enough in the system acquisition process. Software expertise is absent in most program offices. In fact, there is still no way to identify, develop and retain the necessary software expertise. In addition, the lack of adequately defined requirements continues to be the number one complaint of software developers.

CONTINUED

III. Recommendations: The following are areas which should be given careful consideration by the program manager to avoid some of the more obvious software development pitfalls:

1. Address the software aspects of your program head-on and early.

2. Recruit software expertise into the program office early in the acquisition.

3. Insist on a complete, robust, written Software Requirements Specification.

4. Don't skimp on the time allocated to software development.

5. Plan for software and system integration and test before development contracts are awarded.

6. Conduct formal, structured software design reviews.

7. Make sure you get enough of the right documentation.

8. Establish detailed Configuration Management procedures early.

IV. Conclusions: Embedded software is playing an increasingly important role in Department of Defense major system acquisitions. The perception that software development is a "Black Art", and any realities behind the perception, must be changed. Embedded software is far too important to today's defense systems to be left to magic.

Chapter One

INTRODUCTION

BACKGROUND

Computer technology has progressed to the point that every system developed today contains an embedded computer. As our reliance on these computers increases, so too do the challanges for today's program managers. These challanges have grown to the point where embedded software is the cause of major program cost and schedule overruns. In at least one case, the software overrun was equal to the total negotiated system cost. This is not a new problem. The 1976 Charter for the Department of Defense (DoD) Management Steering Committee for Embedded Computer Resources stated:

> Current annual expenditures by the Department of Defense on the design, development, acquisition, management, and operation support of computer resources embedded within and integral to weapons, communications, command and control, and intelligence systems are measured in the millions of dollars. At the same time such computer resources have often presented critical cost and schedule problems during the development and acquisition of new defense systems. Even after system implementation and fielding the software has often proven unreliable. (3:Encl 2)

This has proven to be very frustrating for the program manager, for each of the services and for the DoD as a whole. We are in an age of computers. Household appliances purchased today contain highly reliable computers. Many people have personal computers both at home and at work. Many write their own personal computer programs. What is it about DoD systems that makes software so complex that it can not be costed? Why is DoD software so expensive to maintain? Why can't the DoD get its arms around this problem? How can something which is so straightforward in home application, be the downfall of so many DoD major system acquisitions?

This paper will examine the current realities and illusions of embedded software development. It will provide the program manager with some insight into the special factors which must be considered when embedded software is involved in the acquisition of a major system whose primary purpose is not data automation.

DEFINITION OF TERMS

The mystique of software development is enhanced by the lack of discipline in the use of computer related terminology. The resulting lack of communication makes the task of solving both management and technical problems almost impossible. The following terms, defined by DoD Directive 5000.29, are subject to the most confusion.

Computer Equipment [Computer, Computer Hardware]: Devices capable of accepting and storing computer data, executing a systemic sequence of operations on computer data or producing control outputs. Such devices can perform substantial interpretation, computation, control and other logical functions.

Computer Data [Data]: Basic elements of information used by computer equipment in responding to a computer program.

Computer Firmware: The logical code of computer equipment which interprets the control functions of that equipment.

Computer Program: A series of instructions or statements in a form acceptable to computer equipment, designed to cause the execution of an operation or series of operations. Computer programs include such items as operating systems, assemblers, compilers, interpreters, data management system [system programs], utility programs, and maintenance/diagnostic programs. They also include application programs such as payroll, inventory control, operational flight, strategic, tactical, automatic test, crew simulator, and engineering analysis programs. Computer programs may be general purpose in nature or be designed to satisfy the requirements of a specialized process of a particular user.

Computer Resources: The totality of computer equipment, computer

program, computer data, associated documentation, personnel and supplies.

Computer Software: A combination of associated computer programs and computer data required to enable the computer equipment to perform computational or control functions.

Embedded: Adjective modifier; integral to, from the design, procurement, and operations point of view espoused in DoD Directive 5000.1.

Software Engineering: Science of design, development, implementation, test, evaluation, and maintenance of computer software over its life cycle.(3:Encl 1)

OVERVIEW

This paper will first provide an overview of the software development process and look at common software development problems. It will then identify problems unique to embedded software development and evaluate the DoD Directives and Standards which have application to the management of embedded software development. The paper will conclude with recommendations to be considered by program managers faced with developing and delivering systems in the face of today's challanges.

Chapter Two

THE SOFTWARE DEVELOPMENT PROCESS

INTRODUCTION

Many of the challanges facing today's program manager are related to the common misperception that software development is a "black art". Much of this reputation is undeserved. In order to modify this impression, this chapter will summarize the software development process as it exists independent of the other components of the system to be acquired.

The software development process can be logically divided into six segments: Requirements Definition, Design, Writing the Instructions (Coding), Construction (Integration), Testing, and Documentation. (1:4) Each segment describes a different activity which requires unique skills from the program office, the software developer, and the ultimate user.

REQUIREMENTS DEFINITION

Often left incomplete and frequently even ignored, the activities leading to the complete definition and documentation of software functional requirements must be complete before beginning any other software development activities. This is critical to successful software development. This sounds obvious in theory. In practice, this is the most difficult segment in software development. It requires frequent interaction between the program office, the ultimate software user, and the software developer to describe precisely, in terms all three parties can fully understand, what the software must do and how it must do it -- in a timeframe of great system uncertainty.

The product of Requirements Definition is a document frequently called the Software Requirements Specification (SRS). This document should be able to stand alone in describing the functions and performance of the final software product. It must fully describe the software functionally -- what the software

must do in reaction to which stimulus. In addition, software performance requirements must be defined. These requirements should reflect how often the software will perform each function, how fast, and under what special conditions. The SRS documents the approved baseline of understanding between the parties involved and forms the basis for all other segments of software development.

Software requirements are seldom static throughout the development cycle. (This will be discussed in the next chapter.) The SRS must be a document capable of accommodating change throughout the total software life cycle as the level of understanding of the uses of the software and the implications of various requirements on software design increase. An agreed upon plan to accept and control such change in the requirement's base line, a Configuration Management Plan, is therefore as critical as the SRS itself and must be completed before entering software design.

DESIGN

Once a documented and approved set of software requirements exists, the design segment can logically begin. The purpose of this segment is to identify alternatives for meeting the requirements within the imposed design constraints. These constraints may be any combination of the size and configuration of the computer equipment, the speed with which the software must execute, and the amount of desired or required operator interaction. The selected alternative specifies the computer equipment configuration, the programs which must operate within the software and the interaction between them. Trades are made between "off the shelf" multiple purpose programs which may be purchased directly from a vendor and special purpose programs which must be written to meet either functional or performance requirements.

Roughly akin to an architect's drawing, this segment produces an actual layout of the system hardware and software, to include the programs which must exist within the software and their allocation to computer equipment. It defines the inputs, outputs, and functional and performance characteristics of each hardware and software component as well as the interaction between all computer resources (hardware and software). If necessary, this segment will also include prototype and demonstration of key programs and logic flows.

The design segment directly affects software costs throughout the software life cycle. It drives the complexity of the programs to be written, the success with which those programs will be tested and operated, and the ease with which

programs will be modified to accommodate future requirement changes. The criticality of this segment is reflected in the number of formal reviews required during this activity. There is frequently a Software Design Review (SDR) early in the design activity that demonstrates a clear understanding on the part of the designer of all system requirements. There is almost always a Preliminary Design Review (PDR) midway through the design process which provides an allocation of the software requirements to software components or programs. There is always a Critical Design Review (CDR) which is used to validate, in detail, the total design. Each of these reviews demonstrates increasing levels of detail reflecting a growing understanding of the product to be delivered. Each of these levels of design is formally documented to reflect agreement to the evolving design baseline.

WRITING THE INSTRUCTIONS

Frequently called "coding", this is frequently the most emphasized and yet the easiest segment of software development. Once the software design has been documented and reviewed in detail, the coding becomes an exercise much like working a puzzle -- not necessarily trivial, but more mechanical than creative. It consists of translating the documented logic flow into machine compilable, translatable, or readable language (as a function of the programming language selected and the system programs available).

Coding requires great finesse. No two programmers will produce the same set of instructions for the same logic flow. There are many programs where all possible solutions are often equally correct. This is not true, however, when the program is constrained by either computer size or execution time. In this case the puzzle becomes more difficult and only one of the possible solutions may meet all performance requirements. Computer memory required and time to execute are very sensitive to the choice and order of computer instructions. A poor or missing software design, the selection of an inefficient compiler, or the wrong operating system can make the puzzle impossible to solve and the program impossible to write.

In spite of the complexity of coding, by far the greatest challange in this segment comes in resisting the urge to start too soon -- and the temptations are great. Writing instructions before the design process is complete may save time, but can be extremely risky. Programs today are facing schedule pressures so severe that coding is begun before the requirements are fully documented. This is

proving to be a false economy. Premature coding will place unnecessary constrains on the remaining design or may result in false starts and the necessity to write instructions for the same program twice -- both of which can be expensive in both schedule and budget.

CONSTRUCTION

Once the instructions have been written for each program, the next segment begins, putting the programs together to form their final software product. Commonly called Integration , this segment is frequently left to chance. This segment requires careful planning to ensure the programs will not only fit together, but perform together as they begin to interact in the computer environment. Many programs which perform well in isolation just won't work when placed in the actual operating environment where they must compete for computer resources. Careful construction allows the software design to be validated as the programs come together. This aspect of software development is the source of many system problems. It will be given further consideration in the next chapter.

TEST

The test segment actually begins during coding and continues until actual acceptance of the software. Each program element is thoroughly tested as the instructions are written and formed into subroutines (Unit Test). Each program and combination of programs are tested during software construction (Integration Test). The whole software package is tested again as a Computer Software Configuration Item (CSCI) prior to delivery and operational use. While software design normally takes a "top down" approach, software testing frequently takes a "bottom up" approach. Each test thus considers the smallest possible number of new variables to ensure completeness.

Formal CSCI testing is divided into at least three independent testing cycles. The first test cycle, frequently called Development Test and Evaluation (DT&E), demonstrates the software works using predefined test data. This cycle allows the software paths to be controlled with known inputs and therefore predictable outputs. The second test cycle, frequently called Operational Test and Evaluation (OT&E), demonstrates the software works using actual data. This cycle allows the software paths to be tested using the data of the real world which frequently is different from the more sterile test data. The third test cycle, Operational

Acceptance Test and Evaluation (OAT&E), is conducted by the software system operators demonstrating both that the software works in the operational environment, and that operator training and procedures are adequate. At the conclusion of the third cycle the software is considered ready for operational use. As with construction, successful testing requires careful planning going back as far as the Requirements Definition.

DOCUMENTATION

Documentation is a thorn in the side of every program manager. Done properly, the documentation provides not only a valuable development tool, but will greatly enhance the use of the software and expedite the future modification of the software. Ideally, the documentation segment is concurrent to the other five software development segments.

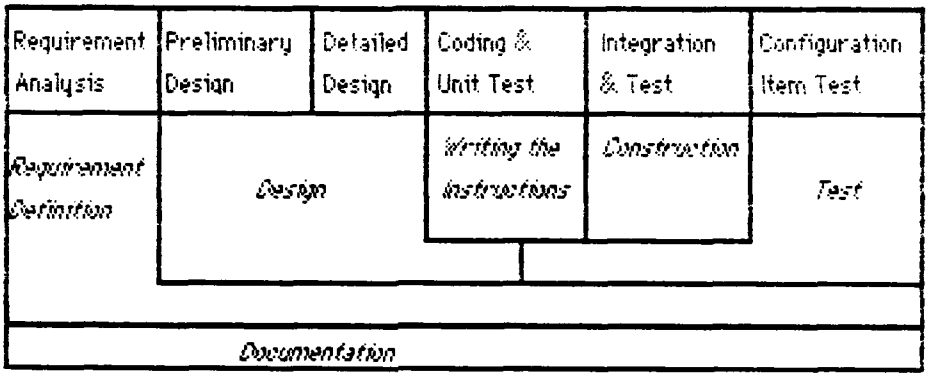
In today's development environment, which is frequently both cost and schedule constrained, documentation falls by the wayside. The decision tends to be made to accept a software product that works and agree to accept the documentation later -- a later which seldom comes. The decision not to complete inadequate or missing documentation is made easier by the fact that such documentation written after the fact tends to have little utility. Much documentation is now a direct translation of the written instructions, written by someone brought in after both the designers and coders have moved on to other projects.

Timely documentation is worth the time and money it takes. It must however be appropriate to the software development activity. The use of each piece of documentation must be carefully considered to ensure it is written at a level of detail appropriate to its purpose. Too often today the same software documentation is written to aid the development process, train system operators and support software maintenance. It tends to be inadequate to meet any purpose in attempting to be general purpose.

Appropriate software documentation can be invaluable in the management of the development process. It can, and should, represent and fix the various development baselines in the development process. It provides an opportunity for widespread review and corrections during the software development process.

PHASES OF SOFTWARE DEVELOPMENT

The software development process may be partitioned in many ways. DoD Standard 2167 divides the software development cycle into six consecutive phases: Software Requirements Analysis, Preliminary Design, Detailed Design, Coding and Unit Testing, Computer Software Component Integration, and Computer Software Configuration Item Testing. Each of these phases is defined by a formal review and a set of approval documentation. The six segments described herein cut across the six DoD phases as shown in figure 1.



THE SOFTWARE DEVELOPMENT PROCESS
FIGURE 1

SUMMARY

The software development process consists of multiple segments executing simultaneously, each contributing to an orderly, phased software development cycle. The six segments of the software development process described in this chapter are unique and require careful consideration by the program manager. This process, while not trivial, seems straightforward enough. Unfortunately, the process seldom operates as advertised; and frequently for reasons out of the control of the program manager. The following chapter will examine some of these reasons.

Chapter Three

COMMON SOFTWARE PROBLEMS

INTRODUCTION

In spite of a logical, disciplined software development process, something always seems to go awry. There are three major areas of difficulty: requirements are seldom well understood and never static; computer software is technically complex; and computer software is abstract and therefore difficult to manage. Each of these areas will be examined in detail in this chapter.

REQUIREMENTS

Software requirements have proven to be very difficult to specify with any precision or accuracy. Most Software Requirements Specifications today seem to be a conglomeration of good ideas, each documented at a different level depending on the maturity of the idea. This results in a hodge-podge ranging from over specification at the implementation level, to under specification of fuzzy concepts. There seem to be at least four reasons for this phenomena: we don't really know what we want, we change our minds about what we want, we don't really know the best way to accurately express what we want, and we are under real pressure to get on with the development.

Describing software requirements seems analogous to shopping for a new tie. You know you want a tie. You even know the suit you want to wear it with. Beyond a vague discussion of colors, you know no more than that you will recognize it when you see it. It is only if you have actually seen the tie you want that you can describe the tie with any precision -- enough precision that you could ask someone else to buy it for you. Software is much the same. How often has the user been asked to describe the volume of message traffic five or ten years into the future? Such volumes are critical to successful software design. The program manager is left with a real dilemma. Such requirements must be anticipated by the program manager much as by the shopper who doesn't know that you don't like paisley ties.

Change in software requirements during the development process is therefore inevitable -- and frequently disasterous. As the design review process progresses the picture of what the software will look like becomes progressively clearer. These reviews may not make what the software should do any clearer, but they will rule out those things that the software does that it shouldn't. (That paisley tie would be ruled out during the design phase.) Simultaneously, as the time of operational use gets closer, the operational environment becomes better defined, and the functional and performance requirements tend to change in response.

Software is falsely perceived as being flexible and therefore amenable to frequent changes. This myth has been perpetuated by creative software developers who deliver apparent miracles overnight. These "miracles" are very confusing to those not involved in developing software. There is an inverse rule that seems to apply. In software, if the change is large, it can be accommodated easily; if the change is small, it is almost impossible to implement. The complexity of any change is a function of the stage of development in which it is introduced and its compatibility with the existing design. The program office faced with changing the length of a single data base item used thoughout the software may have to redesign its software -- essentially start all over again.

Even when we know what we want, we can not express the requirement precisely enough that this desire is obvious to all those who use the requirements document. There is no commonly understood language of "computerese" which allows such requirements to be expressed in a disciplined way. Basic language differences between the user, the developer, and the program office introduce confusion in the statement of each requirement. The user will describe the requirements in operational terms, the program office states requirements in acquisition terms, and the developer attempts to translate both into computer related terms. The confusion resulting from this process leads to a seemingly endless chain of technical interchange meetings between the three parties and a series of verbal agreements. The danger of verbal agreements speaks for itself.

In the meantime, the program office is under considerable pressure from both within and without to "get on with it". It is almost impossible to measure the progress of the requirements definition phase -- therefore difficult to evaluate the value of spending another six months, six days or even six hours prior to the start of the design phase. The user is eager to see a product come in the door, the program office is committed to a schedule and a budget profile, and the developer is eager to get on contract and to get people to work.

It falls to the program manager to identify when requirements definition has achieved its goals of providing sufficient understanding of the software product to allow for productive design. It also falls to the program manager to provide for a continuation of the requirement definition activity throughout the development process to allow for the orderly processing of the inevitable changes to the requirements baseline.

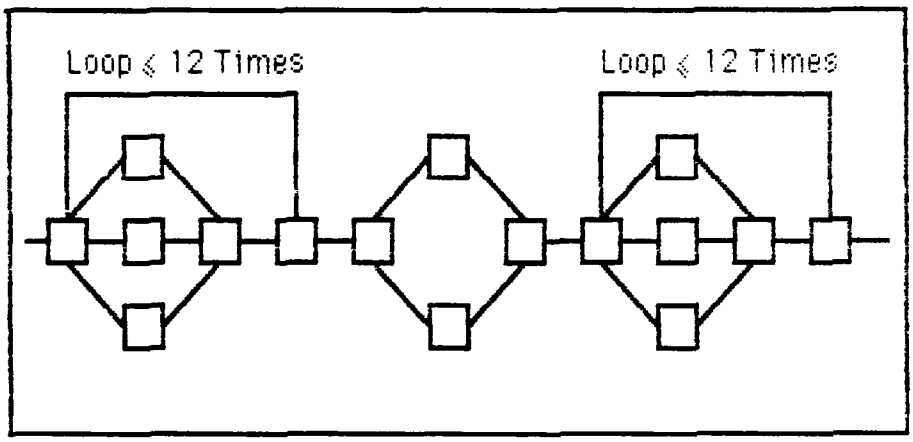
SOFTWARE IS COMPLEX

Computer software is both technically and logically complex. It is technically complex in the sense we are asking computers to solve problems that have never been solved before; the problems themselves are complex. Computer software is logically complex in the sense that computer software consists of multiple computer programs, each of which potentially contains thousands of instructions. The affect of software complexity is frequently underestimated; especially in today's world where so many functions are being performed by computers.

Ten to twenty years ago, the types of problems solvable by a computer were limited by the state of computer technology. Computers were very big, required special care, and were very expensive. Computer equipment technology has now progressed to the point that the software created twenty years ago will operate in portable computers no larger than cigarette packages for a fraction of the cost. Today, the types of problems which can be solved by a computer are limited only by imagination. We are allocating problems to computers which have not been solved any other way. "Weather forcasting, nuclear equations, orbital calculations, gravity effects, ballistics -- all of these require special mathematical, engineering, or scientific skills or knowledge."(1: 69)

The inherent technical complexity of today's problems is compounded by the demands made on computer performance. Many functions require complex software to operate in "real time", decisions must be made almost immediately (where almost is defined in nanoseconds). This requires great precision in the use of computer time. Many functions require complex functions to operate in computers severely constrained by size or capability. This requires great precision in the use of computer hardware resources. Many functions require complex software to operate in an environment which is both equipment and performance constrained. (Today's advanced avionics systems are good examples.). These programs are very complex to design, code, and test.

Computer software is logically complex due to the magnitude of combinations and permutations of the instructions in all but the simplist of programs. This makes testing and verification all but impossible. "People who state they want 100 percent error-free software are talking about very small programs -- or do not know what they are talking about."(1:70) Figure 2 shows a logic flow developed for a RAND Corporation Study. This flow contains only 3 decisions, two of which are repeated 12 times before moving on.



LOGIC FLOWCHART
FIGURE 2

> Even through this simple flowchart, the number of different paths is about ten to the twentieth. If one had a computer that could check out one path per nanosecond (10^{-9} seconds) and had started to check out the program at the beginning of the Christian era (1 A.D.), the job would be about half done at the present time. (4:4)

Complexity of both types has a direct affect on the cost, schedule, and risk associated with software development. It must not be underestimated. Careful management of all aspects of complexity is required for successful software development.

SOFTWARE IS ABSTRACT

Software presents a unique management challenge. "It is managed and created 'in the blind'." (1:12) It has defied all attempts to develop reliable means for costing or scheduling. There are no intermediate milestones which provide insight into potential program success. After delivery there is still nothing that can be touched or picked up, or even pointed to. The software either works or it doesn't. From the point of view of the program manager this can be very frustrating and intimidating:

> You software guys are too much like the weavers in the story about the Emperor and his new clothes. When I go out to check on a software development, the answers I get sound like,"We're fantastically busy weaving this magic cloth. Just wait awhile and it'll look terrific." But there's nothing I can see or touch, no numbers I can relate to, no way to pick up signals that things aren't really all that great. And there are too many people I know who have come out at the end wearing a bunch of expensive rags or nothing at all. (11:1)

Much of this reputation is undeserved. Software appears abstract because it is not well understood. This lack of understanding is encouraged because managers of software projects tend to come from non-software disciplines. They tend to be afraid of the "black art" of software development. There is little hope for real change in this respect for the next five to ten years though there are small signs of improvement. "Software people" are not being groomed for software management responsibility. The opposite is more true. They are rewarded for being unmanageable. Recognized, successful software experts are inclined to be non-conformist whiz kids.

Both the DoD and the computer industry as a whole are fighting today to make software more manageable. Industry's solution seems to be to separate computer software from other engineering disciplines so that it can be given direct management attention. DoD's solution is to introduce more formality and discipline into the software development process (see Chapter Five). Neither solution has proven completely successful. Software development is still too young. The limits of software development have not yet been identified; creativity is still required. More important, the talent necessary for successful software management has not yet been identified. We must continue to focus on the

"problems" of managing software development until these basic issues have been resolved, not until then can we begin to train and grow the necessary talent.

SUMMARY

There are many things that can, and do, go wrong in software developement. Too frequently we charge off without a good appreciation of where we are going and how we are going to get there. The lack of accurately defined requirements is a frequent complaint. The inherent complexity of each software development is consistently underrated -- especially as it affects cost and schedule. The aura of magic that has traditionally surrounded software development has made this the rule rather than the exception. Each of these problems are magnified when the software development is just a piece of a larger acquisition -- when a system whose primary purpose is not data automation contains software. This will be the subject of the next chapter.

Chapter Four

EMBEDDED SOFTWARE DEVELOPMENT

INTRODUCTION

Software plays an ever expanding role in today's major system acquistion. Due to a recent history of software failures system program managers view this with some alarm. As a result, there is a tendency to put off software decisions until all of the other system decisions have been made. While this mode of decision making does limit the number of software decisions to be made at the system level, it can be catastrophic to the software development process. A Rand study observes: "Perhaps the most important observation [of the study] is that software, despite its mission critical nature, is not routinely treated as a major decision variable from the earliest stages of the acquisition process." (10:3)

Early system level decisions will strongly affect the complexity of the software to be developed, the flexibility of the software in response to system or operational changes, and the visibility of the program manager into the software development process. No single approach to embedded software development will guarantee success to every program manager. The approach must vary based on the type of the system and the maturity of the hardware and software technologies involved as well as the need for future system growth. There are, however, some areas of consideration for any major system acquisition containing embedded software. This chapter will look at four considerations: hardware/software trades, software interface definition, acquisition strategy development, and system integration.

HARDWARE/SOFTWARE TRADES

Today's technology provides realistic trades between hardware and software capabilities for many system problems. Current technology has introduced many oportunities for the substitution of computer solutions for many traditionally

mechanical problems. For example, a wrist watch will tell good time independent of whether it relies on a computer or on mechanical gears to operate. Program managers, given such a choice, have historically chosen the mechanical watch. There is no magic about how it works or what it takes to build or maintain one. Today the optimum solution is frequently the computer driven watch. You can no longer open the watch and see it work, yet the technology has been demonstrated and such a watch has proven to be cheaper and more reliable. In addition, the computer watch can do more than the traditional mechanical watch. It can tell time with more precision, tell the date, and frequently also operate as a calculator. With the accelerated advances in computer technology, computers are capable of assuming more nontraditional functions cheaply and reliably. Much of this technology is not as commonplace as the wrist watch; much of it is yet to be proven in functions as complex as Defense systems.

Program managers are still opting for the mechanical solution where it exists. Every program manager must actively seek out all opportunities for system trades; to include hardware/software trades. Informed trades between hardware and software implementations can make the difference between developing a system which must be replaced in five years (the current useful life for much command and control software) or a system capable of accommodating the unexpected modifications to which all systems fall prey.

DEFINITION OF INTERFACES

Software is inherently flexible. This flexibility has been used as the excuse for carelessly defined software intefaces based on the assumption that the software development can take up the slack. Poorly defined interfaces, however, can severely limit the inherent flexibility. The importance of carefully considered and defined interfaces must not be underestimated in delivering software within cost and schedule. Each system design decision, even some which appear to be unrelated, can have a significant impact on the potential software design.

Today embedded software is a critical component in the evolving system design. Too frequently, the software interfaces are dictated by default once the other elements of the system have been defined rather by a balanced consideration of the whole design. The software designer is too often left attempting to place the proverbial ten pounds in a five pound bag. The system interfaces have a direct influence on the ultimate flexibility of the software design and therefore its ability to accommodate the inevitable changes in system and operational requirements. It also has a direct affect on the complexity and, therefore, cost,

schedule, and risk of the software development. Improperly defined interfaces or overconstrained performance requirements in support of those interfaces can make successful software delivery virtually impossible.

ACQUISITION STRATEGY

The unique talents required for embedded software development add a new complexity to the traditional development of a system acquisition strategy. The decision whether to contract directly or rely on a system prime contractor for the software is not an easy one -- and there is no decision that is right for all cases. This decision, however, determines who identifies and controls the software requirements and interfaces. It also determines the System Program Office (SPO) visibility into the software development costs and schedule. As embedded software assumes larger and more critical roles in our major systems the decision becomes more difficult.

The opportunity to describe software which is small and totally contained within a segment is becoming more rare. Today's computer supported control systems must have access to all aspects of the executing system. We are now developing networks of computers where a computer supporting one function is capable of filling in for a computer primarily intended to support a much different task. Computer software now interfaces to more system components than ever before -- and this trend is likely to continue.

Can a system program manager delegate the management of embedded software to the contractor selected because of expertise in some other area? This decision is not unique to software. It is more difficult, however, because good software managers are so hard to come by. On the other hand, the decision to contract directly requires the program manager to have a trained staff capable of managing the software integration. Can the program manager recruit and keep this talent throughout the system development? Can the program afford not to?

There are cases where the magnitude of the software is considered to be of sufficient magnitude that a contract is let independent of other system segments. If the size of the software development is estimated to require approximately three years for development (not an excessively long period in software development terms), the contract must be negotiated before the hardware contractors may have reached preliminary design. This has resulted in contracts without a specification of the software interfaces -- which can not be defined for as long as a year. The program manager is left over a barrel.

There are successes and failures to prove every case. It is only by carefully balancing all of the acquisition decisions, with careful consideration of all aspects of the system, that the program manager can hope to develop an acquisition strategy to support successful system development.

SYSTEM INTEGRATION

Just when it seems the challanges have been met and the problems overcome, the pieces of the system start to come together -- and the system doesn't work. Too often the acquisition strategy is developed without full consideration of how segments of the system will be tied together contractually and operationally. Planning for the integration of embedded software is not to be taken lightly. The system integration will surface problems that aren't apparent in any other stage of development. Consideration of system integration and contract close out must begin in the earliest planning phases of the acquisition process and continue through the design and development phases of system acquisiton.

The inherent complexity of embedded software makes this problem more difficult. Unanticipated combinations and permutations of both inputs and outputs can bring the embedded software to its knees. Because of the logical complexity of any software, such combinations and permutations almost always occur. Each interface can be thoroughly tested, yet the system still won't work. Integration planning can make the difference between taking the system apart and starting all over again or being able to isolate and correct the faults as they occur.

SUMMARY

The fact that software is still considered and managed as a "black art" by many program mangers causes the problems of embedded software development to be compounded. Yesterday's approach of ignoring the software until it has been described by the other elements of the system just won't work any more. Embedded software is capable of assuming an even wider range of activities than ever before and is taking on larger and larger roles in major system acquisition. Embedded software must be given consideration as an active decision variable throughout all phases of the acquisition process, with special emphasis early in system acquisition planning.

Chapter Five

DOD DIRECTIVES AND STANDARDS

INTRODUCTION

The Department of Defense has been working to come to terms with the difficulties of embedded software development. On 4 June 1985, the Departmant of Defense published DoD Standard 2167, a new standard for "Defense System Software Development". Simultaneously, Military Standards 483, 490, and 1521 were substantially revised. The effort to provide program managers with a solid foundation of standards designed to avoid the kind of horror stories heard in the past about embedded software continues. The current standards provide a disciplined approach to software development. This chapter will examine the Standards and the appropriate DoD Directives as they support the program manager in embedded software development.

DOD DIRECTIVES

DoD Directive 5000.1, Major System Acquisitions (29 March 1982)

This directive describes the Defense Systems Acquisition Review Council (DSARC) and the approval and review process for major system acquisition new starts. It also describes the role and responsibilities of the program manager in "acquiring and fielding (in accordance with instructions from line authority) a system that meets the approved mission need and achieves the established cost, schedule, readiness and affordability objectives." (2:11)

The rules of engagement for the program manager in the battle for program approval are established by this directive. It does not, however, provide any specific guidance for acquisitions which contain embedded software. It is mentioned here only because the directives that follow are amplifications of this basic directive.

DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems (26 April 1976)

This directive specifically addresses the acquisition of major systems containing embedded software. It describes requirements for computer resource (hardware and software) review as part of the formal DSARC major system acquisition review process. This directive requires validation of computer resource requirements and a Computer Resources Life Cycle Plan prior to DSARC II, the approval for full scale development. It directs that computer resources, both hardware and software, be specified and treated as configuration items and directs the use of Higher Order Programming Languages whenever feasible.

The Management Committee for Embedded Computer Resources charter is contained in this directive. This committee has four objectives: improve computer resource management, increase top level visibility in computer resource development, form a coordinated technology base program, and guide the assimilation and integration of computer resource policy, practice, procedure, and technology into the normal process of major defense systems acquisition.

This directive has been under revision since before June of 1983. It requires the formal consideration of embedded software before the start of full scale development -- a point relatively late in the acquisition process. It follows both Concept Exploration and Demonstration and Validation. As discussed in the previous chapter, software must be treated as an active decision variable from the earliest phases of the acquisition process. This directive provides the program manager with a check list of those facets of the software development that DSARC II will be looking for, but does not provide the insight the program manager will need to ensure that embedded software is given appropriate attention before this milestone is reached.

DoD Directive 5000.31, Computer Language Policy (Draft, 10 June 1983)

This directive specifies the use of the Ada programming language for all DoD mission critical applications. The utility software, operating systems, and compilers necessary for widespread application of this directive are still under development at this writing. Sufficient progress has been made that the Ada language can now be used for applications on specific machines, but has not yet been accepted as an industry standard.

The use of a common DoD language has significant long term implications and potential cost savings, but Ada is still a new language. It will take a few years before a trained labor pool and sufficient utilities exist to make use of this language for embedded software truely cost effective.

DOD STANDARDS

DoD Standard 2167, Defense System Software Development (4 June 1985)

This is a new standard which was written to directly address the problems of developing mission critical computer software. It divides the software development process into six consecutive phases (Software Requirements Analysis, Preliminary Design, Detailed Design, Code and Unit Testing, Computer Software Component Integration and Testing, and Computer Software Configuration Item Testing). It relates these six phases to the ongoing system level development activities, but focuses on formalizing the activities, documentation, and review process for each of the six software development phases.

This standard describes a " by the numbers" approach to software development. It provides an extremely formal and rigid approach which addresses not only the formal deliverables and review cycles, but also describes the contractor's internal activities in support of each phase. This provides a two edged sword. The program manager now has a standard which provides a reliable, logical approach to software development. With strict adherence, however, the software is now much more expensive. The standard requires the documentation and formalization of internal contractor procedures, both of which tend to be very expensive.

Followed blindly, application of this standard may provide the program manager with a false sense of security. the standard does not replace software management expertise in the program office. Such standards in the past have caused contractors to focus on the delivery of documentation, frequently documentation that was generated independent of the software development itself.

This standard will save much effort in redefining the software development process for each software acquisition. It must be used with much care, however, as many software applications are not amenable to this rigid software development process.

Military Standard 483A, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs (4 June 1985)

A lack of appropriate configuration management has been the downfall of many a software development. This standard describes the phases in software development where formal baselines are appropriate, and provides guidelines for auditing, maintaining, and controlling each of these base lines. It provides specified procedures for controlling the different types of system components to include control of system interfaces and computer software requirements.

Military Standard 490A, Specification Practices (4 June 1985)

This standard provides guidelines for the preparation of all levels of system specification. It provides a generalized system specification tree which demonstrates the relationship of the various specifications generated during system development, and provides an outline and guidelines for each specification in the tree. This standard reinforces DoD Standard 2167 in that it expands the definition of each of the specifications required throughout the software development process. It provides specific references to the appropriate Data Item Descriptions (DID) for each software specification required to document each baseline in the development process.

Military Standard 1521B, Technical Reviews and Audits for Systems, Equipments, and Computer Programs (4 June 1985)

This standard provides detailed guidance on the description, format, and major subjects for review for each of the formal reviews in the system development process. This standard reinforces DoD Standard 2167 in expanding the description of the review processes themselves for each of the major program milestones.

SUMMARY

The Department of Defense has provided guidelines, which if used correctly, result in a solid foundation for the management of mission critical software developments. These guidelines take full advantage of the best of current software development and management practice. However, these guidelines must be carefully considered in the light of the specific needs of the embedded software under acquisition. These standards applied without full consideration of

the system specific implications will result in significantly higher costs for software development. This consideration should take special account of the maturity of the software technology to be applied (small self-contained software developments will not require the same rigor as large software developments with many critical system interfaces), the life cycle plan (if all software changes will be accomplished by the operating agency, the documentation requirements may be more rigid than if most change will occur under development contract), and the Operations Concept (documentation written to assist the operator is different from documentation to assist a software maintenance activity).

No directive or specification can or should relieve the the program manager of the responsibility for careful thought and planning for all aspects of the system acquisition. The DoD directives and standards discussed here provide a solid foundation upon which to base this consideration for mission critical or embedded software development.

Chapter Six

CONCLUSION

RECOMMENDATIONS

The increasing dependency of DoD systems on computer technology adds significantly to the complexity of major system acquisition. The discipline of software development is relatively new -- and is proving to be very different from more traditional engineering disciplines. This is not to say, however, that the presence of embedded software adds more complexity to program management than does today's budget crunch or micro-management from congress. The following are areas which should be given careful consideration by the program manager in order to avoid some of the more obvious software development pitfalls.

Address the software aspects of your program head-on and early.

Up front consideration of the implications of software complexity on all system level decisions from the earliest phases of system acquisition is the single best software development risk reducer. Don't let yourself be intimidated into postponing critical software decisions until the rest of the system has been fully explored. Thoroughly considered software interfaces and performance requirements will simplify design, coding, and test. Don't forget to look past the development process and consider the system operations and maintenance in these critical early decisions.

Recruit software expertise into the program office early in the acquisition.

Software engineering talent has not yet been well identified, developed or exploited. Studies are underway today to identify the specific training and characteristics required to support major system acquisition. Expertise in data automation or programming is not enough. You must find someone who is knowledgeable about software but able to maintain a system perspective. Without this talent inside the program office you will be totally dependent on the expertise of your contractor(s) -- a position of some risk.

Insist on a complete, robust, written Software Requirements Specification.

The lack of a solid, approved SRS is the most frequently heard complaint from software developers. The perception is that the government just can't get it's act together. There have been too many occasions where the SRS is written after the software has been delivered. Today's most common solution is to ask the development contractor to write the specification. The pressure is thus on a contractor to tell the government what the government wants to buy -- often from the same contractor who writes the specification. While this has obvious drawbacks, it must be carefully considered -- program offices seldom have the resources or the expertise necessary to write such a document.

Don't skimp on time allocated to software development.

There is a very real tendency to underestimate the complexity of the software to be developed -- and to therefore underestimate the total cost and/or schedule allocated to the development itself. It is very difficult for anyone to understand why one piece of software can cost so much more than another piece and even harder to understand why it takes so long to write code in support of a relatively simple function. Resist! Software development seldom goes much faster with the addition of people to the activity -- and management problems increase. Time and money spent early in the process will pay for itself in the system life cycle.

Plan for software and system integration and test before development contracts are awarded.

As with so many activities in life, if you can see the end of the project at the beginning, you are more than half way there. Too many development contracts are let without careful consideration of what will be required to integrate, test, demonstrate and deliver a fully coherent system. This has proved disastrous. Poor up front planning costs both budget and schedule late in the development. DoD Std 2167 provides useful insights as to where in the development process meaningful planning can be initiated and updated.

Conduct formal, structured software design reviews.

Software design reviews are used today to provide software system overviews. This has resulted in cases where the government sends more people to

the review than the contractor has working on the design. It is extremely difficult to conduct a meaningful review with a room full of people -- many of which are only there to find out what the software is supposed to do rather than critique how it will do it. This is not the intent of Mil Std 1521B. Insist that any review material is distributed in sufficient time for considered analysis before the review. Insist that all participants involved in the review are well prepared. Insist that the format for the review is understood and followed. Above all, insist that your objectives are met by the review -- if necessary schedule separate meetings to meet other valid objectives.

Make sure you get enough of the right documentation.

Documentation is the single most abused area of software development. Consider carefully the specifications of Mil Std 490A. Balance this consideration with the utility of each document for your system -- and with your program offices ability to credibly review this documentation for suitability for these uses. Documentation is expensive because most software developers do not like to write it. You can not afford not to have the visibility into software development activities afforded by good documentation -- nor can you afford the cost of documentation which will not be read.

Establish detailed Configuration Management procedures early.

Mil Std 483A is frequently forgotten. Too many program offices struggle with the attempt to define anew the means of managing and controlling system level changes, configuration management. Change during the software development process is inevitable -- plan for it early and don't let it get away from you. Uncontrolled changes will bring a software development to it's knees.

CONCLUSION

Embedded software is playing an increasingly important role in DoD major system acquisitions. Yet program managers still hold the wide spread belief that software is a "black art" created by a strange breed of person called a programmer. This perception, and any realities behind the perception, must be changed. Software is far too important to today's defense systems to be left to magic.

The need to bring software out of the closet has been recognized both within the software industry and the Department of Defense. Both are working to create an environment for software development where software can be reliably costed,

scheduled and produced in response to well defined requirements. While there have been successes in this endeavor, there is still a long way to go.

The Department of Defense has been aggressively addressing the problems of managing a major system which has a high dependence on software development. DoD Standard 2167 responds to the need to remove the mystique from software development by documenting a disciplined, uniform approach to DoD software development. This standard will remove much of the management uncertainty surrounding software development by providing clear milestones and well defined reviews to certify each milestone.

There are many problems still to be solved. There is still not an accepted approach for costing of software and the current climate makes it very difficult to budget for uncertainties. There is still a lack of needed software engineering talent to support the system program offices in the early stages of concept development. It is impossible to test every software contingency -- yet when we are talking about software embedded in our major defense systems it is unclear how much testing is enough.

Software development is a relatively new discipline. It will take time before it has achieved the kind of maturity comparable to other development disciplines. We, as program managers, must remain sufficiently flexible as we, together with industry, assend the learning curve.

BIBLIOGRAPHY

Books

1. Fox, Joseph M. Software and Its Development. Englewood Cliffs, NJ: Prentiss-Hall, Inc., 1982.

Official Documents

2. DoD Directive 5000.1. Major System Acquisitions. 29 March 1982.

3. DoD Directive 5000.29. Management of Computer Resources in Major Defense Systems. 26 April 1976.

4. DoD Directive 5000.31. Computer Programming Language Policy. Draft, 10 June 1983.

5. DoD Standard 2167. Defense System Software Development. 4 June 1985.

6. Military Standard 483A. Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs. 4 June 1985.

7. Military Standard 490A. Specification Practices. 4 June 1985.

8. Military Standard 1521B (USAF). Technical Reviews and Audits for Systems, Equipments, and Computer Programs. 4 June 1985.

Unpublished Materials

9. Carlson, Raymond A., Maj, USAF and Massman, Thomas A., Maj, USAF. "Software Development for Weapon Systems". Research study prepared at the Air Command and Staff College, Air University, Maxwell Air Force Base, AL, May 1974.

CONTINUED

10. Glaseman, Steven and Davis, Malcolm. Software Requirements of Embedded Computers: A Preliminary Report. Santa Monica, CA, The Rand Corporation, Report R-2567-AF, March 1980.

11. Rand Corporation. Software and Its Impact: A Quantitative Assessment. Santa Monica, CA: Report 4947, December 1972.

END

DTIC

5-86